Program Verification in a Logical Theory of Constructions

Peter Dybjer

Programming Methodology Group, CTH, S-412 96 Göteborg, Sweden

Abstract: The logical theory of constructions is a simple theory which combines functional programs and intuitionistic predicate calculus. Here we propose that it is a practical alternative to other constructive programming logics, such as Martin-Löf's type theory. Its main advantage is that it admits reasoning directly about general recursion, while maintaining that all typed programs terminate. We illustrate the use of this theory by verifying the general recursive subtractive division program.

1. Introduction

The intuitionistic theory of types presented in Martin-Löf (1982) is intended to be a formalisation of constructive mathematics. Suitably interpreted it may also be used as a formal system for reasoning about terminating functional programs. It is thus a logic of total correctness. It has been used by several authors, for example Nordström (1981).

Martin-Löf's type theory is a general framework for introducing types and justifying proof rules for judgements of the four forms A *type*, $A = B$, $a \varepsilon A$, and $a = b \varepsilon A$, by appealing to the semantics given in Martin-Löf (1982). It is in particular possible to interpret propositions as types so that the judgement $a \varepsilon A$ is read as "a is a proof object for the proposition A". The proof rules at the end of Martin-Löf (1982) are given with this reading in mind. We refer to this specific collection of rules of type theory as TT79 so as to keep the distinction between this and the general framework of type theory clear.

When using TT79 as a programming logic and thus giving $a \varepsilon A$ the reading "a is a program for the task (specification) A" two problems appear: the absence of proof rules for general recursion; and the sometimes unavoidable presence of unwanted constructions in the synthesised programs.

The first problem can be tackled without extending TT79. As Smith (1983) demonstrated one can synthesise programs which use primitive recursion of higher type which compute the same functions and behave in a similar way as the desired general recursive programs. This can be done for large classes of general recursive programs, for example the ones which are provably total in first-order arithmetic.

Another possible way of coping with general recursion was suggested by Paulson (1984) who added a rule of well-founded induction to TT79.

The second problem appears in the following context. Consider the introduction rule for the Σ-type:

$$\frac{a \in A \qquad b \in B(a)}{\langle a,b\rangle \in (\Sigma x \in A)B(x)}.$$

The Σ-type is used to interpret existential propositions: we wish to show that there exists a program of type A, which satisfies a property B (more precisely, the property interpreted as the following family of types: $B(x)$ *type* $(x \in A)$). To do this we construct a program $a \in A$ and a proof-object $b \in B(a)$. But usually we are only interested in the program a, and not in the particular proof-object b which we have constructed to show that a satisfies the desired property B. Therefore Nordström and Petersson (1983) suggested introducing a new type former, the subset type former, which can be used to interpret existential propositions. It has the following introduction rule:

$$\frac{a \in A \qquad b \in B(a)}{a \in \{x \in A \mid B(x)\}}.$$

Thus both problems can be dealt with in a more or less satisfactory way, but the underlying principle of identifying programs and proof-objects is in question.

Therefore we shall look at an alternative formalisation of constructive mathematics: the logical theory of constructions, abbreviated LTC, which was formulated by Smith (1978) and which has its origins in first-order theories of combinators in which Aczel (1977) interpreted early versions of Martin-Löf's type theory. An extended version of LTC was formulated by Smith (1984). The extensions are needed to interpret the whole of TT79, including universes, in LTC. They are not needed for the example we present here, however.

LTC has the same basic collection of programs as TT79 (see appendix) (this excludes those programs of the extended LTC which encode propositions and those of TT79 which encode types), but is not based on the identifications of propositions and types and of programs and proof-objects. Instead it is based on ordinary intuitionistic first-order predicate calculus. Its basic predicates are the convertibility relation $a = b$ and the natural number predicate $N(a)$. It has a small and simple collection of inference rules.

One reason for chosing LTC, and not some other theory which combines functional programs and predicate calculus, is that its relation to TT79 is clear: Smith (1984) has shown how to translate TT79 into it, and we thereby know that each program derivation in TT79 can be translated into a program derivation in

LTC. So we know that LTC is at least as strong as TT79. Still, it could of course be the case that program proofs are generally shorter or more natural in TT79. However, we believe the opposite since we can reason about general recursion directly, as we show below, and since we do not identify programs and proof-objects.

We first give most of the inference rules of LTC from Smith (1978) but use a different syntax. Then we introduce the fixed point operator as an abbreviation and the fixed point property as a derived conversion rule. Finally, we give the proof of the subtractive integer division program.

2. Inference rules of LTC

We use the same notation for the basic program forms as, for example, Nordström, Petersson and Smith (1985). However, we use a left-associative infix * instead of *app* for function application. In this notation the inference rules of LTC given by Smith (1978) are as follows: (The only difference is that we only show the rules for the constructors *true* and *false* instead of for the whole collection m_n and thus only for the selector *if* _ *then* _ *else* instead of for the whole collection R_n. We also use the theory of expressions and arities and the convention of identifying classes and properties by putting $a \in A \equiv A(a)$ from Martin-Löf (1983), see appendix.)

Conversion rules

CB: $\mathit{if}\ \mathit{true}\ \mathit{then}\ a\ \mathit{else}\ a' = a$

CB: $\mathit{if}\ \mathit{false}\ \mathit{then}\ a\ \mathit{else}\ a' = a'$

CN: $\mathit{rec}(0,a,e) = a$

CN: $\mathit{rec}(s(m),a,e) = e(m,\mathit{rec}(m,a,e))$

$C\times$: $\mathit{split}(\langle a,b\rangle,e) = e(a,b)$

$C+$: $\mathit{when}(\mathit{inl}(a),d,e) = d(a)$

$C+$: $\mathit{when}(\mathit{inr}(b),d,e) = e(b)$

$C{=}{>}$: $\lambda(b)*a = b(a)$

Reflexivity

$=I$: $a = a$

Substitution

$$=E:\quad \frac{a = b \qquad C(a)}{C(b)}$$

Rules for natural numbers

$$NI:\quad N(0) \qquad\qquad \frac{N(m)}{N(s(m))}$$

$$NE:\quad \frac{N(n) \qquad C(0) \qquad \begin{matrix}(N(x), C(x))\\ C(s(x))\end{matrix}}{C(n)}$$

Logical rules

$$\&I:\quad \frac{P \qquad Q}{P \;\&\; Q}$$

$$\&E:\quad \frac{P \;\&\; Q}{P} \qquad\qquad \frac{P \;\&\; Q}{Q}$$

$$\vee I:\quad \frac{P}{P \vee Q} \qquad\qquad \frac{Q}{P \vee Q}$$

$$\vee E:\quad \frac{P \vee Q \qquad \begin{matrix}(P)\\ R\end{matrix} \qquad \begin{matrix}(Q)\\ R\end{matrix}}{R}$$

$$\to I:\quad \frac{\begin{matrix}(P)\\ Q\end{matrix}}{P \to Q}$$

$$\to E:\quad \frac{P \to Q \qquad P}{Q}$$

$$FE:\quad \frac{FALSE}{P}$$

$$\exists I:\quad \frac{B(a)}{(\exists x)B(x)}$$

$$\exists E:\quad \frac{(\exists x)B(x) \qquad \begin{matrix}(B(x))\\ P\end{matrix}}{P}$$

(restriction: x must neither occur free in P nor in any assumption on which the upper occurrence of P depends other than $B(x)$.)

$$\forall I:\quad \frac{B(x)}{(\forall x)B(x)}$$

(restriction: x must not occur free in any assumption on which $B(x)$ depends.)

$$\forall E\colon \frac{(\forall x)B(x)}{B(a)}$$

LTC has no rule of extensionality. This is not very significant. For example, if we wish to show that two programs f and g which map natural numbers into natural numbers are extensionally equal as functions on natural numbers, i.e. $f = g \in N \to N$ in type theory, then we just have to show that

$$(\forall x \in N)(f{*}x = g{*}x).$$

This idea can of course be iterated if the range is of higher type.

3. Types and termination

The intended models of LTC are the Frege structures of Aczel (1980). This semantics does not give us direct information about termination properties in the way that the semantics of type theory in Martin-Löf (1982) does. In type theory it is the case that if A is a type and $a \in A$, then it is immediately clear that a has all the termination properties we want (see below). So types only contain terminating programs in type theory. (Observe that this is different from the types of the functional language ML, which contain non-terminating programs as well.)

In type theory a type is defined by saying what its canonical elements (programs whose outermost forms are constructors) are (and when two canonical elements are equal, but let us forget this here). An element of a type is one whose value (under =>, see appendix) is a canonical element of the type.

Consider for example the natural numbers. From the above we see that the following three rules:

- 0 is a natural number;
- if a is a natural number, then $s(a)$ is a natural number;
- if $a \Rightarrow b$ and b is a natural number, then a is a natural number;

are the introductory clauses of an inductive definition of the natural numbers in type theory. From this definition it is obvious that if a program belongs to the type of natural numbers, then it terminates both under => and under the full evaluation relation ==> (which corresponds more closely to what we usually mean by termination, see appendix).

This kind of argument is valid for other type formers as well and Martin-Löf has suggested calling a program a, for which $a \in A$ *for some type* A in type theory, *hereditarily terminating* (w.r.t. A), see the discussion after Martin-Löf

(1984).

We shall now show that we can prove termination in LTC in a similar way. First define

$$b \in BOOL \equiv b = true \vee b = false$$
$$q \in A{+}B \equiv (\exists x \in A)(q = inl(x)) \vee (\exists y \in B)(q = inr(y))$$
$$p \in A{\times}B \equiv (\exists x \in A)(\exists y \in B)(p = \langle x,y\rangle)$$
$$f \in A \nrightarrow B \equiv (\exists b)((\forall x)(x \in A \to b(x) \in B) \;\&\; f = \lambda(b))$$
$$p \in (\Sigma x \in A)B(x) \equiv (\exists x \in A)(\exists y \in B(x))(p = \langle x,y\rangle)$$
$$f \in (\Pi x \in A)B(x) \equiv (\exists b)((\forall x)(x \in A \to b(x) \in B(x)) \;\&\; f = \lambda(b))$$

where the bounded quantifiers are defined by:

$$(\exists x \in A)B(x) \equiv (\exists x)(A(x) \;\&\; B(x))$$
$$(\forall x \in A)B(x) \equiv (\forall x)(A(x) \to B(x)).$$

If A is a property which can be written down using only N, $BOOL$, $+$, $\times$, $\nrightarrow$, Σ, Π, then call A an *LTC-type*. From the definitions we can derive inference rules for the LTC-types and the bounded quantifiers.

Our intention is to show that if A is an LTC-type and we can prove $a \in A$ in LTC, then a has all the termination properties we want. To do this, we show that if A' is the type in the sense of type theory which corresponds to A (for example, N' is the type of natural numbers as defined above; $BOOL'$ is the type which has the canonical elements *true* and *false*; etc.), then $a \in A$ is provable in LTC implies that $a \in A'$ is provable in type theory. This justifies saying that if $a \in A$ is provable in LTC, then a is hereditarily terminating w.r.t. A.

Smith's translation of types of TT79 into LTC is essentially translating A' into A as defined above. Moreover, programs are translated into themselves (which is why we write $a \in A'$ instead of $a' \in A'$). Smith showed by induction on the length of the derivation in TT79 that, if we can prove $a \in A'$ in TT79, then we can prove $a \in A$ in LTC. The reverse implication, which we need here, does unfortunately not hold (consider the case where a uses general recursion). So we need to appeal directly to the semantics of type theory in order to get the following theorem.

Theorem: Let A be an LTC-type and let A' be the corresponding type in the sense of type theory. Then we can prove $a \in A$ in LTC iff we can prove $a \in A'$ by appealing to the semantics of A' in type theory.

Proof: The proof is by induction on the structure of A. We show only one base case, $A \equiv N$, and one step case, $A \equiv B \twoheadrightarrow C$, here.

$A \equiv N$: Assume $a \in N'$. This can be deduced by finitely many applications of the three rules for forming natural numbers in type theory above. Since $a \Rightarrow b$ implies that $a = b$ in LTC, a corresponding deduction can be carried out in LTC by using the rule of substitution instead of the third rule.

For the converse, assume $a \in N$. We prove that this implies $a \in N'$ by giving LTC a domain interpretation as follows: a program a is interpreted as the filter $[\![a]\!]$ of formal neighbourhoods as in Martin-Löf (1983D); $a = b$ is interpreted as $[\![a]\!]$ and $[\![b]\!]$ are equal filters; N is interpreted as the set of principal filters $[\![N]\!] = \{\uparrow 0, \uparrow s(0), \uparrow s^2(0), ..\}$; logical constants are interpreted as themselves. The rules of LTC are valid under this interpretation. Martin-Löf (1983D) showed the validity of the conversion rules, for example. Thus $a \in N$ implies $[\![a]\!] \in [\![N]\!]$, and from the definition of principal filters we see that $[\![a]\!] = \uparrow s^n(0)$ implies $a \Rightarrow s(a_1)$, $a_1 \Rightarrow s(a_2)$, ..., $a_{n-1} \Rightarrow s(a_n)$, $a_n \Rightarrow 0$. Thus $a \in N'$.

$A \equiv B \twoheadrightarrow C$: The induction hypothesis is that the theorem is proved for B and C. Assume $f \in (B \twoheadrightarrow C)'$. Then for some b, $f \Rightarrow \lambda(b)$ and $x \in B'$ implies $b(x) \in C'$. Thus $f = \lambda(b)$ in LTC, and $x \in B$ implies $x \in B'$ implies $b(x) \in C'$ implies $x \in C$ (by applying the induction hypothesis twice). Hence $f \in B \twoheadrightarrow C$ by the definition of $B \twoheadrightarrow C$.

For the converse, assume $f \in B \twoheadrightarrow C$, Then for some b, $f = \lambda(b)$ and $x \in B$ implies $b(x) \in C$. By appealing to the domain interpretation again, we have that $[\![f]\!] = [\![\lambda(b)]\!]$ and hence for some d, $f \Rightarrow \lambda(d)$ and $[\![d]\!] = [\![b]\!]$. It follows that $x \in B'$ implies $x \in B$ implies $b(x) \in C$ implies $d(x) \in C$ implies $d(x) \in C'$ (by applying the induction hypothesis twice). Hence $f \in (B \twoheadrightarrow C)'$. $\square$

4. Verifying a general recursive program in LTC

The main reason for looking at LTC is that it can be used for reasoning about general recursion. So we define the fixed point operator in the usual way:

$$fix(f) \equiv ((\lambda x)f(x*x))*((\lambda x)f(x*x)).$$

We can thus derive the conversion rule:

$$Cfix\colon fix(f) = f(fix(f)).$$

We wish to verify the following subtractive division program:

$$div(i,j) \equiv divh*i*j$$
$$divh \equiv fix((g)(\lambda i)(\lambda j)if\ lt(i,j)\ then\ 0\ else\ s(g*(i-j)*j))$$

What we wish to show is that *div* satisfies the division property, i.e.,

$$(\forall i \in N)(\forall j \in N)(j > 0 \rightarrow DIV(i,j,div(i,j)))$$

where

$$DIV(i,j,q) \equiv (\exists r \in N)(r < j \ \&\ i = j\times q+r)\ \&\ q \in N$$
$$i < j \equiv lt(i,j) = true$$
$$j > i \equiv i < j$$
$$i \geq j \equiv lt(i,j) = false.$$

(See appendix for the undefined forms.) Observe that we need to make the termination condition of $div(i,j)$ explicit by requiring $div(i,j) \in N$ (the second conjunction of the *DIV*-proposition).

The proof below uses the derived rule of well-founded induction on natural numbers:

$$WE: \ \frac{n \in N \qquad \dfrac{((\forall y \in N)(y < x \rightarrow C(y)))}{C(x)}}{C(n)}.$$

This rule is easily derived from ordinary induction on natural numbers.

In the proof below we write

Ass: *A*

...

Con: *B*

to represent the hypothetical proof of the conclusion *B* from the assumption *A*, and use indentation to indicate that the intermediate inferences depend on the assumption *A*. The proof is completely formal except that we have not derived all the basic arithmetic facts that we use from first principles. The parts where such derivations would be filled in, if we were to make the proof completely formal, are indicated by dots. Otherwise we indicate in the left margin which proof rule has been used. Note that $\forall NI$ and $\forall NE$ refer to the derived introduction and elimination rules for the bounded quantifiers $\forall a \in N$.

```
         Ass: i ε N
           Ass: j ε N
             Ass: j > 0
               Ass: (∀i' ε N)(i' < i → DIV(i',j,div(i',j)))
                 ...
                 i < j ∨ i ≥ j
                 Ass: i < j
                   ...
(below)            div(i,j) = 0
NI                 0 ε N
=E                 div(i,j) ε N
                   ...
                   i = j×div(i,j)+i
&I                 i < j & i = j×div(i,j)+i
∃I                 (∃r)(r < j & i = j×div(i,j)+r)
&I               Con: DIV(i,j,div(i,j))
                 Ass: i ≥ j
                   ...
                   i-j ε N
∀NE                i-j < i → DIV(i-j,j,div(i-j,j))
                   ...
                   i-j < i
→E                 DIV(i-j,j,div(i-j,j))
&E                 div(i-j,j) ε N
NI                 s(div(i-j,j)) ε N
                   ...
(below)            div(i,j) = s(div(i-j,j))
=E                 div(i,j) ε N
&E                 (∃r)(r < j & i-j = j×div(i-j,j)+r)
                   Ass: r < j & i-j = j×div(i-j,j)+r
&E                   r < j
&E                   i-j = j×div(i-j,j)+r
                     ...
                     i = j×s(div(i-j,j)+r
=E                   i = j×div(i,j)+r
&I                   r < j & i = j×div(i,j)+r
∃I                 Con:(∃r)(r < j & i = j×div(i,j)+r)
∃E                 (∃r)(r < j & i = j×div(i,j)+r)
&I               Con: DIV(i,j,div(i,j))
∨E             Con: DIV(i,j,div(i,j))
WE           Con: DIV(i,j,div(i,j))
→I         Con: j > 0 → DIV(i,j,div(i,j))
∀NI      Con: (∀j ε N)(j > 0 → DIV(i,j,div(i,j)))
∀NI (∀i ε N)(∀j ε N)(j > 0 → DIV(i,j,div(i,j))).
```

In order to show how the fixed point rule is used, we shall fill in the details of the second omitted proof fragment, i.e. prove that the following inference is valid:

$$\frac{i \in N \quad j \in N \quad i < j}{div(i,j) = 0}$$

or after replacing *div* and < by their definiens:

$$\frac{i \in N \quad j \in N \quad lt(i,j) = true}{fix(\tau)*i*j = 0}$$

where

$$\tau(g) \equiv (\lambda i)(\lambda j) if\ lt(i,j)\ then\ 0\ else\ s(g*(i-j)*j).$$

The proof of this is as follows:

$$\begin{array}{ll}
 & Ass\colon i \in N,\ j \in N,\ lt(i,j) = true \\
CB & if\ true\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) = 0 \\
{=}E & if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) = 0 \\
C{\to} & \tau(fix(\tau))*i = (\lambda j) if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) \\
C{\to} & \tau(fix(\tau))*i*j = if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) \\
{=}E & \tau(fix(\tau))*i*j = 0 \\
Cfix & \tau(fix(\tau)) = fix(\tau) \\
{=}E & Con\colon fix(\tau)*i*j = 0.
\end{array}$$

This is part of the base case. Let us also show the corresponding part of the induction step:

$$\frac{i \in N \quad j \in N \quad i > j}{div(i,j) = s(div(i-j,j))}$$

or after replacing *div* and > by their definiens:

$$\frac{i \in N \quad j \in N \quad lt(i,j) = false}{fix(\tau)*i*j = s(fix(\tau)*(i-j)*j)}.$$

The proof of this is as follows:

$$\begin{array}{ll}
 & Ass\colon i \in N,\ j \in N,\ lt(i,j) = false \\
CB & if\ false\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) = s(fix(\tau)*(i-j)*j) \\
{=}E & if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) = s(fix(\tau)*(i-j)*j) \\
C{\to} & \tau(fix(\tau))*i = (\lambda j) if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) \\
C{\to} & \tau(fix(\tau))*i*j = if\ lt(i,j)\ then\ 0\ else\ s(fix(\tau)*(i-j)*j) \\
{=}E & \tau(fix(\tau))*i*j = s(fix(\tau)*(i-j)*j) \\
Cfix & \tau(fix(\tau)) = fix(\tau) \\
{=}E & Con\colon fix(\tau)*i*j = s(fix(\tau)*(i-j)*j).
\end{array}$$

5. Conclusion

Our intention has been to present a system of proof rules for reasoning about functional programs which is close in spirit to TT79, but which is simpler and, at least in some respects, more convenient to use.

We have already explained the differences between TT79 and LTC, let us summarise the similarities:

- They are both theories about the same collection of functional programs. These functional programs are evaluated in normal order.
- They both contain natural deduction proof rules for intuitionistic first-order predicate calculus. These rules are expressed differently though. In LTC they are expressed directly, whereas in TT79 they are expressed via the

interpretation of propositions as types.

— They are both logics of total correctness and use the fact that $a \in A$ implies that a terminates. They are not intended to be used for reasoning about programs which compute partial functions. (This could presumably be done indirectly, for example through codings of domain theory.)

We have only discussed the relationship between LTC and Martin-Löf's type theory here. We hope that this discussion will be relevant also when trying to understand similar aspects of other logics for reasoning about functional programs, such as Constable's (1982) intensional version of type theory, Coquand and Huet's (1984) impredicative theory of constructions, Manna and Waldinger's (1980) deductive synthesis (compare for example their proof of the general recursive subtractive division program) and Cartwright and McCarthy's (1979) first-order programming logic. Also Scott's, and Milner, Morris and Newey's (1975) LCF must be mentioned here, even though it is different in important respects: it is a logic for reasoning about partial as well as total objects and contains proof rules for inequality as well as for equality.

Finally, a word about synthesis versus verification. It may seem that LTC does not support synthesis of a program from its specification in the same way as TT79 or deductive synthesis (both of these identify specifications and propositions and derive programs from the proofs of the propositions). In the proof above the program was given before the proof and it may seem that this must necessarily be the case. But no, given the specification, there is no reason why we have to know the structure of the program until we wish to apply an inference rule which appeals to this structure. If we thus decide on the structure of the program only when we are forced to do so, then we could carry out a process which is similar to that which we could carry out in TT79 or deductive synthesis.

References

P. Aczel, The strength of Martin-Löf's type theory with one universe, *Proceedings of the Symposium on Mathematical Logic*, Oulu, 1974, Report No 2, Department of Philosophy, University of Helsinki (1977) 1-32.

P. Aczel, Frege structures and the notions of proposition, truth and set, in *The Kleene Symposium* (North-Holland, 1980) 31-59.

R. Cartwright and J. McCarthy, First order programming logic, in *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, San Antonio (1979).

R. Constable, Intensional analysis of functions and types, internal report CSR-118-82, Department of Computer Science, University of Edinburgh (1982).

T. Coquand and G. Huet, A theory of constructions, preliminary version presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (1984).

Z. Manna and R. Waldinger, A deductive approach to program synthesis, *ACM TOPLAS*, 2 (1)(1980) 92-121.

P. Martin-Löf, Constructive mathematics and computer programming, in *Logic, Methodology and Philosophy of Science VI, 1979* (North-Holland, 1982) 153-175. Also in *Mathematical Logic and Programming Languages*, (Prentice-Hall, 1984).

P. Martin-Löf, unpublished notes from a series of lectures given in Siena (1983).

P. Martin-Löf, The domain interpretation of type theory, unpublished notes from a lecture given at the Workshop on Semantics of Programming Languages, Göteborg (1983D).

R. Milner, L. Morris, M. Newey, A logic for computable functions with reflexive and polymorphic types, in *Proceedings Conference on Proving and Improving Programs*, Arc-et-Senans (1975).

B. Nordström, Programming in constructive set theory: some examples, in *Proceedings of the 1981 Conference on Functional Languages and Computer Architecture*, Portsmouth, N.H. (1981) 141-154.

B. Nordström and K. Petersson, Types and specifications, in *Information Processing 83* (North-Holland, 1983) 915-920.

B. Nordström, K. Petersson, J. Smith, *An Introduction to Martin-Löf's Type Theory*, in preparation (1985).

L. Paulson, Constructing recursion operators in intuitionistic type theory, technical report no.57, University of Cambridge Computer Laboratory (1984).

J. Smith, On the relation between a type theoretic and a logical formulation of the theory of constructions, Ph.D. thesis, Department of Mathematics, University of Göteborg (1978).

J. Smith, The identification of propositions and types in Martin-Löf's type theory: a programming example, in *Foundations of Computation Theory*, LNCS **158** (Springer-Verlag, 1983) 445-456.

J. Smith, An interpretation of Martin-Löf's type theory in a type-free theory of propositions, *Journal of Symbolic Logic* **49** (3) (1984) 730-753.

Appendix A.

Expressions, arities and definitional equality: (see Martin-Löf (1983) for a full account).

Arities are built up from the ground arity (), and in general have the form $(a_1 \dots a_n)$ where n is a natural number and $a_1, \dots, a_n$ are arities. Each expression e has a certain arity a, which we write

$$e - a.$$

Expressions are built up from constants and variables by means of application and abstraction. Assume that we have the constant (or variable) $f - (a_1 \dots a_n)$ and the expressions $e_1 - a_1, \dots , e_n - a_n$. Then we can form the *application*

$$f(e_1, \dots, e_n) - ().$$

Moreover, assume that we have the expression $e - ()$ and the variables $x_1 - a_1, \dots , x_n - a_n$. Then we can form the *abstraction*

$$(x_1, \dots, x_n)e - (a_1 \dots a_n).$$

(We may also use certain syntactic conventions, such as infix and mixfix notation and writing $(\lambda x)e$ for $\lambda((x)e)$, $(\forall x)e$ for $\forall((x)e)$, etc.)

Certain expressions are *definitionally equal* written $\equiv$, for example,

$$((x_1, \dots, x_n)e)(e_1, \dots, e_n) \equiv e[e_1, \dots, e_n/x_1, \dots, x_n].$$

Abbreviations can be introduced by stating that certain expressions are definitionally equal. If we have the expression $e - ()$ and the variables $x_1 - a_1, \dots , x_n - a_n$, then we can introduce the constant $c - (a_1 \dots a_n)$ by

$$c(x_1, \dots, x_n) \equiv e.$$

In this paper we are concerned with the following program forms which are common to LTC and TT79:

Constructors:

$true$ - ()
$false$ - ()
0 - ()
s - (())
$\langle\_,\_\rangle$ - (()())
inl - (())
inr - (())
λ - ((())).

Selectors:

$if\ \_\ then\ \_\ else\ \_$ - (()()())
rec - (()()(()()))
$split$ - (()(()()))
$when$ - (()(())(()))
$\_*\_$ - (()()).

We use the following abbreviations:

$m+n \equiv rec(n,m,(n',y)s(y))$
$m\times n \equiv rec(n,0,(n',y)y+n)$
$m-n \equiv rec(n,m,(n',y)p(y))$
$p(n) \equiv rec(n,0,(n',y)n')$
$lt(m,n) \equiv rec(n,false,(n',y)or(y,eq(m,n')))$
$eq(m,n) \equiv eqi(n)*m$
$eqi(n) \equiv rec(n,(\lambda m)zero(m),(n',y)(\lambda m)rec(m,false,(m',z)y*m'))$
$zero(n) \equiv rec(n,true,(n',y)false)$
$or(a,b) \equiv if\ a\ then\ true\ else\ b.$

Computation rules:

$$\frac{b \Rightarrow true \quad a \Rightarrow c}{if\ b\ then\ a\ else\ a' \Rightarrow c}$$

$$\frac{b \Rightarrow false \quad a' \Rightarrow c}{if\ b\ then\ a\ else\ a' \Rightarrow c}$$

$$\frac{n \Rightarrow 0 \quad a \Rightarrow c}{rec(n,a,e) \Rightarrow c}$$

$$\frac{n \Rightarrow s(m) \qquad e(m,rec(m,a,e)) \Rightarrow c}{rec(n,a,e) \Rightarrow c}$$

$$\frac{p \Rightarrow \langle a,b \rangle \qquad e(a,b) \Rightarrow c}{split(p,e) \Rightarrow c}$$

$$\frac{q \Rightarrow inl(a) \qquad d(a) \Rightarrow c}{when(q,d,e) \Rightarrow c}$$

$$\frac{q \Rightarrow inr(b) \qquad e(b) \Rightarrow c}{when(q,d,e) \Rightarrow c}$$

$$\frac{f \Rightarrow \lambda(b) \qquad b(a) \Rightarrow c}{f*a \Rightarrow c}.$$

Moreover, a canonical program (one the outermost form of which is a constructor) has itself as value.

The computation rules define a certain value relation $\Rightarrow$. This is not the usual one (to *fully evaluated form* in the terminology of Nordström, Petersson and Smith (1985)), which we denote $\Rrightarrow$ and which is obtained from $\Rightarrow$ by continuing to evaluate the parts of arity () of a canonical program. For example,

$$s(0)+s(0) \Rightarrow s(s(0)+0)$$

but

$$s(0)+s(0) \Rrightarrow s(s(0)).$$